

Python unittest

作者：王琦

创建日期：2017 年 5 月 15 日

最后修改：2017 年 5 月 15 日

我们可以针对特定的函数进行单元测试。单元测试应该为该函数提供参数，并观察输出结果是否符合预期。单元测试可以是黑盒的，它不需要去假设函数的内部实现，只考察输出结果是否符合预期，这样在函数、系统发生重构时，单元测试仍可以发挥作用，检查新的实现是否符合业务逻辑。

单元测试具有如下特性：

- test fixture: 通过 `setUp()` 和 `tearDown()` 函数完成 执行测试前的准备工作 和 执行测试后的清理工作
- test case: 测试用例
- test suite: 定制测试的执行顺序（本文不予讨论）
- test runner: 测试报告生成器，可以是 GUI 或 CLI（本文不予讨论）

1. 基本例子

1.1 编写单元测试

在 `day4/workspace` 中存在文件 `test_string.py`：

1. 首先，我们定义一个继承自 `unittest.TestCase` 的类 `TestStringMethods`。（这个类的名字是任意的）
2. 其次，我们在这个类中定义它的成员函数，每个成员函数负责测试一个功能点。注意到，这些成员函数都需要以 `test_` 开头，这样 `Test Discovery` 才可以找到、执行这些函数。

- [官方文档 The list of assert methods](#) 提供了用于测试的标准库函数：如 `assertEqual()` 等。

3. 在 `main()` 函数中执行 `unittest.main()`

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

1.2 执行单元测试

a) 通过上述步骤，我们就写好了测试字符串的模块。在同级目录的命令行下，我们可以进行如下操作：

```
python test_string.py
```

命令行将输出：

```
...
-----
Ran 3 tests in 0.000s

OK
```

b) 我们也可以给命令加上 `-v` (verbose) 选项，输出更多的细节信息：

```
python test_string.py -v
```

命令行将输出：

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok
```

```
-----
Ran 3 tests in 0.001s
```

```
OK
```

2. 命令行工具 及 测试搜索器

我们可以不在 `if __name__ == '__main__':` 中编写 `unittest.main()`, 相反, 我们可以通过另一种命令来执行单元测试。目前, 在 `day4/workspace` 目录下有 4 个文件:

```
workspace
|---- test_string
|       |---- test_upper(self)
|       |---- test_isupper(self)
|       |---- test_split(self)
|---- test_foo
|       |---- test_foo(self)
|---- test_bar
|       |---- test_bar(self)
|---- test_fixture
|       |---- ...
```

请执行下述命令, 观察并理解其输出结果:

```
// 启用 unittest 模块, 并只执行 test_foo 文件中的单元测试
python -m unittest test_foo

// 只执行 test_foo 模块中 TestFoo1 类包含的单元测试
python -m unittest test_foo.TestFoo1

// 只执行 test_string 模块中 TestStringMethods 类的 test_upper 包含的单元测试
python -m unittest test_string.TestStringMethods.test_upper

// 命令等价于 python -m unittest discover, 搜索并执行当前目录下所有的单元测试
python -m unittest
```

3. 组织测试代码

我们可以单元测试类 (我们定义的 `TestCase` 的子类) 中重写 `setUp()` 和 `tearDown()` 函数, 来定制单元测试执行前后的准备和清理策略。 `test_fixture.py` 提供了一个简单的例子:

```
from unittest import TestCase

class TestFixture(TestCase):

    def setUp(self):
        print('Do preparation work here.')

    def test_nothing(self):
        pass

    def tearDown(self):
        print('Do clean work here.')
```

由于一些测试是在一定前提条件下才能进行的，所以 `text fixutre` 特性的应用十分广泛。

a) 以 GUI 为例，判断一个窗口的大小或是其他控件的状态，需要先创建窗口（请求资源）；同时，在测试结束时，需要关闭窗口（释放资源）。[官方文档 26.4.4. Organizing test code](#) 提供了这样的一个例子。

b) 对于一个 Web 系统的后端服务，我们想要测试一些函数、算法是否正确，但这些函数要求数据库中存在有数据。那么我们便可以在 `setUp()` 函数中调用数据库接口，创建数据；待测试完成后，再 `tearDown()` 函数中删除这些数据。